March 2008

Mobile Shopping Assistant

Berk Birand
Worcester Polytechnic Institute

MOBILE SHOPPING ASSISTANT

A Major Qualifying Project

Submitted to the Faculty

of the

Worcester Polytechnic Institute

In partial fulfillment of the requirements for the

Degree of Bachelor of Science in

Computer Science

by

Berk Birand

Date: March 18, 2008

APPROVED:

Professor Emmanuel O. Agu, Project Advisor

Professor Murali Mani, Project Co-Advisor

Abstract

Due to information overload, many people currently struggle with coordinating and managing their basic chores. By using wireless localization, we have designed and developed the "Personal Assistant," a location-based program that facilitates grocery shopping. The "Personal Assistant" helps the user find the best deals in grocery items, and filters the information based on the stores' locations. It is a proof-of-concept application that utilizes several cutting-edge technologies to investigate how they can be used together.

Acknowledgments

Although it has one author, this project is not the result of individual effort, but rather the product of concerted teamwork. Without the help of my advisors Prof. Emmanuel Agu and Prof. Murali Mani, this project could not have matured passed its infancy. Both were profoundly involved at every stage of the development, from inception to fruition.

I would like to thank my parents, Serda and Refik, for their endless trust and for awarding me with an excellent education, and my brother, Burak, for continuing the engineering practice in the family.

Contents

1 Introduction

1.1 Information overload in the 21st century

In today's fast paced world, it is becoming increasingly hard to keep track of the simplest chores. As people are more and more involved in their work-related activities, many people do not have any time for the simpler tasks. Many employees find themselves working overtime and answering hundreds of e-mails a day, at the cost of regularly visiting their drive-through fast-food place.

In order to deal with this overload of information, we should use the very same processing power that made our lives harder in the first place: computers. By delegating some of the tasks that used to be our brain's responsibility to the computer, we can use our minds to do more productive endeavors.

This project attempts to do just that, by delegating the chores related to grocery shopping. Although seemingly trivial, shopping for groceries is becoming increasingly complex on very different levels. One needs to keep track of the items to be purchased, where they can be bought for the best prices. Since the large supermarkets are more concerned about overall profits than item-based profit, some items can be sold at significantly discounted prices. Continually watching for these deals is a very time-consuming task, but it can potentially save a considerable amount of money.

1.2 Proposed solution

The solution that we are proposing is that of having a personal assistant program running on a portable device that will handle some of these basic, shopping-related chores. This version of this software will more specifically deal with shopping for groceries, and finding the best deals for them.

The user is only asked to input his grocery list. The software then takes

care of finding the items at discount prices at the closest stores available. This will delegate yet another task that our brains currently perform to the computers. The user will not have to worry about where to buy the items, and can instead use his mental powers for more creative purposes.

1.3 Scenarios

Scenario 1 Peter is away from home for a business meeting. On his way to the meeting, he receives an e-mail notifying him that the meeting has been postponed by an hour. Being in a foreign location, Peter wants to buy some grocery items that he needs. He takes out his PDA, and runs the Personal Assistant program. This in turn looks up for all discount stores and supermarkets in the area, and guides him to the closest one that carries the items that he is looking for. Not only does he buy all the items that he needs, but he also manages to find the stores in a foreign location.

Scenario 2 When looking at a grocery store's discount specials page, Mary notices that the flank steak that she buys regularly is available at half the price for only a week. Mary decide to become a smarter shopper and track down deals like these at local stores. Yet she quickly realizes that it is virtually impossible to keep track of all the discounts in the area, as there are tens of stores, and the deals usually expire within a week. She installs the Personal Assistant program and inputs the items that she buys on a regular basis. When she has time to shop for groceries, she simply uses her PDA to find the best deals without doing any preliminary research.

1.4 Project Goals

While computing has become ubiquitous, the processing power of computers have been used for various purposes. In a typical college curriculum, a

student takes a multitude of classes in different fields including algorithms, computer networks, databases and human-computer interaction. Although all of these areas seem independent in their own rights, their usefulness is apparent only when they are used together to form an application that accomplishes more than the sum of its parts. There is no point in studying databases and SQL if the data will not be utilized. The study of algorithms is a very exciting field, but implementing them just as an exercise is a superficial approach. Their advantage becomes more apparent when used for solving a real world problem. With the computing ecosystem so diversified, the question remains as to how well these technologies work together. How do they help the cause of computing by improving society as a whole?

The goal of this project is to investigate how each of these modern technologies work together, while creating the personal shopping assistant. We will create a proof-of-concept application that uses as many different technologies, while delivering a service to the regular user. This service would be very hard, if not impossible, to procure by only utilizing one system. The service that was picked was that of helping the user with part of the daily chores, notably that of grocery shopping.

2 Literature Review

Since one of the purposes of the project is to merge a lot of smaller subsystems into a larger system, it makes sense to first look into each of these systems in their own rights. This section introduces the many buildings blocks used by this system.

2.1 Wireless Localization

Although wireless localizations technologies are a recent invention, they are quickly finding their way into many mainstream devices. Most recently, the iPhone has added wireless localization to its services[1].

2.1.1 History

Localizing the position one the globe has always been a fascination of humans. Throughout history, various techniques were devised to find out where exactly we are located. Some use mechanical devices to measure the angles between the stars and the sun (figure 1) , while others relied on measuring

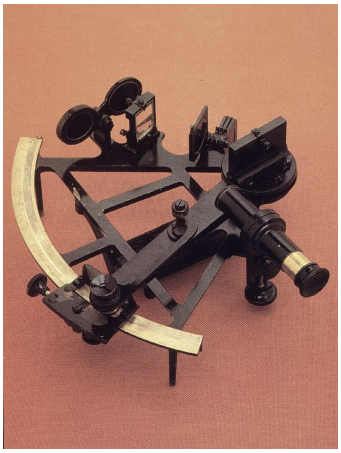

Figure 1: Sextant used for measuring the latitude

[1] http://www.wi-fiplanet.com/news/article.php/3723366

the time of the travel, and estimating the distance traveled from a known location. These two methodologies (absolute positioning on the globe, and positioning relative to a known location) are still the main ones used for determining location, and can be roughly characterized as absolute and relative positioning.

The major breakthrough in terms of absolute positioning was made in the 1960s with the advent of the GPS system. Once the first few satellite launches were successful, it became clear that one could use a series of satellites orbiting around the globe to position oneself. Like the sextants of the old days, modern GPS devices make measurements from several satellites to pinpoint a person's location on the globe. The same time-signal is received from several different sources at once. By measuring the arrival time of each signals, the receiver can calculate its relative position to each satellites. Since the satellites' locations are known, very accurate results can be obtained that are within meters of the actual location.

Relative positioning mechanisms resurfaced through the use of wireless localization. With the advent of cheap wireless access points, many home networks moved on to a wireless infrastructure. This trend filled the major cities with thousands of access points. These access points are mostly secured using encryption mechanisms, and cannot be used by strangers to connect to the Internet. However, they perform the job of marking locations very convenient

2.1.2 Basic mechanism

Wireless localization works by mapping the access points (which have unique identifiers in them, their MAC addresses) to a location. The mechanism works in two phases. First, the mappings should be created, and secondly, they should be looked up.

The first phase of using localization is made possible by performing a variant of war-driving. In war-driving, people drive around the city while

scanning for wireless networks. The aim is to locate access points that are not protected with any encryption scheme, and can therefore be exploited for obtaining free access to the Internet. For localization purposes, the aim is not to hunt for non-encrypted network, but rather to find out their coordinates. The scanning is done by computers that have both wireless cards, and GPS devices mounted on them. When a new access point is encountered, the wireless card reads its signal strength and other characteristic values (such as its Service set identifier (SSID) and its MAC address). The GPS receiver is used to obtain the absolute coordinates. Together these two sets of data are stored in a database for later use.

The second phase of localization is used when a user wishes to find out her location. No GPS device is needed for this phase, as all the information that is needed is already present in the database. Just as before, the wireless card scans for the networks in the area. The combination of access points received by the user as well as their signal strengths are used to search the database for the closest match and hence determine the user's location.

However the algorithm is not as trivial as it sounds. Signal strengths can be affected by many other factors. For instance, WiFi signals can usually bounce from walls when indoors, making such calculations harder. Other factors such as the weather, or the presence of other devices working in the same frequency range (such as microwaves) can also affect the quality of the signal. Triangulation algorithms need to take these factors into account when estimating the location.

2.1.3 Comparison of Wi-Fi to GPS

Wireless localization methods are usually compared with GPS because they perform roughly the same task. However, this is not a rightful comparison: the two technologies are not mutually exclusive but they complement each other. They each have their own strengths and weaknesses.

The major problem with GPS that motivated the research in other areas

is its poor indoor coverage. GPS relies on signals received from satellites, which allows it to perform very well when there is a direct line-of-sight to the satellites. This feature makes GPS an excellent choice when used outdoors, and explains why it it has found such acceptance in airplanes and hikers alike. But the reliance on satellites is also the Achilles' heel of the system when being used indoors. The signal strength is very weak, if not absent indoors. Most recent buildings have a steel construction, which makes them act as Faraday's cage, blocking all the electro-magnetic signals.

Wireless localization excels within buildings. The access points are usually placed such that they will have a wide reception, increasing the range of the network connection. The improved range also means improved estimates when it comes to measuring the location.

Another area where the two technologies differ is in relation to their antennas. Most recent devices rely on wireless networks to get online, and therefore have built-in wireless antennas. GPS is often seen as an accessory instead of a necessity; it requires additional hardware, an extra cost that makes the product more expensive without adding much to it. After all, one usually knows where one is, and does not need the device to tell them that.

Last but not least, it is worth mentioning that the coverage of wireless networks is restricted to major cities. In the more rural areas, wireless networks are not available, and therefore GPS remains the only option.

2.1.4 Conclusion

Wireless localization is a very exciting technology that promises to complement GPS. It uses the widely available network of access points to pinpoint one's location on the globe quite accurately, especially indoors.

2.2 Localization libraries

As explained in section 2.1, wireless localization is a very promising technology. Although it is still considered a new technology, there already are a lot of companies and research institutes working on localization using wireless networks. This section lists some of the available libraries that determine coordinate estimates of locations in order to pick one that will be used for this project.

2.2.1 Skyhook Wireless

SkyHook Wireless is a company that develops wireless localization software. They have built their own access point location databases for looking up coordinate information. The majority of the employees at Skyhook are drivers that have cars equipped with special computers. These computers have wireless cards, as well as GPS receivers, and they are used for coming up with the necessary mappings. They boast of covering 70% of the United States' population, sub-second time-to-fix, 99% indoor availability and 10-20m accuracy.

The dynamic coverage map found on their web site seems to cover the entirety of Worcester and the greater Boston area. Although Skyhook is mainly focused on WiFi, they also have a technology called the "Hybrid Positioning System" that can combine other location sensing systems, such as GPS, IP location, Cell towers and WiMax to provide better estimates.

They provide a Software Development Kit (SDK) free for development purposes (but charges when product launches commercially). The SDK can be used for academic research as well.

SkyHook also publishes a specific online API called loki.com for building location-sensitive web pages. The user installs a browser plug-in that performs the database lookup to determine the user's location. The location information can then be sent to web pages supporting this technology. Those sites can then know the user's location and modify their content appropri-

ately.

Testing For experimenting with the system, we decided to run the sample application that comes bundled with SkyHook's SDK. However, we could not get that application to begin with. The compilation failed at the linking stage, and was related to a version mismatch with the *libcurl* library that was available on our Linux system. Since it is a proprietary system, we were unable to find the error by looking at the source code. Our attempts at finding help on their support groups were unsuccessful. We posted a message, but the help that we got was not very helpful. The follow-up messages that we sent remained unanswered.

SkyHook was never a preference for this project anyway, because it is not open-source and it is written in C++. Although a very important language historically, C++ does not facilitate quick development cycles. Debugging is not very easy, and manual memory management makes tracking errors very hard.

2.2.2 PlaceLab

PlaceLab is an open source library for creating location-sensitive applications. It is developed mainly by researchers at the University of Washington and the Intel Research lab at Seattle. It essentially performs the same job as Skyhook, but uses a public database instead of holding its own. While Skyhook is mainly oriented for WLAN localization, PlaceLab supports a variety of other mechanisms (such as GPS and Bluetooth).

The access point database they use is called wiggle.net, and is also a public web site. Anybody having the required hardware can post their access point information to the web site.

PlaceLab is therefore a very good low-cost alternative to Skyhook. It has some shortcomings too. For instance, there is virtually no support present, as the mailing lists are pretty much unused. The documentation is very

minimal, and the most in-depth questions are left unanswered. On the plus side, being an open-source software written in Java, some questions can be resolved by browsing the source code itself (which is not that well documented either).

There is a hardware compatibility list on their website, and apparently it does not work with all kinds of hardware. There are also several sample applications that have been created as part of a group project for a graduate level Computer Science class at the University of Washington.

Testing Installing PlaceLab and running the sample applications weren't trivial. The details of these are explained in appendix A. It is worth mentioning that the documentation that came with PlaceLab is rudimentary at best, and fails to explain more fine-tuned configuration options. We had to delve into the source-code for troubleshooting a lot of the problems we ran into.

PlaceLab downloads a list of the beacons in a given state from *wiggle.net* and saves it in a local cache. Most of the access points (APs) around the WPI campus were within the coverage range of the set of beacons. Yet the APs inside the buildings cannot be discovered. The absence of the WPI access points may be due to security concerns on the part of WPI's Network Operations. Without any reference APs, we cannot get coordinate estimates inside buildings. We therefore needed some sort of an estimate for demonstration purposes, and we came up with two options for doing this :

1. add the access point inside the building to the main Wiggle.net database by filling out the data entry form,

2. find a way to "reverse engineer" the local cache, and insert the access point information only in the local cache.

For security purposes, we did not want to expose the WPI access points to the public, so we decided to use the second option, namely adding the ac-

cess points and their locations to the local cache. We found that the cache was stored in a local database system called HSQLDB. This database system is used offline from a desktop application, and can be queried through the standard JDBC interface. The queries can be performed using SQL, which allows for executing complex queries efficiently. We managed to find a program that will allow us to run SQL queries on the local database where HSQLDB stores its information. After discovering the structure of the tables, we added the appropriate records in the database, which then managed to display the estimates even inside the building where we were doing the demonstration.

2.2.3 Other Libraries

There are a few other libraries that perform similar jobs, although they are not very relevant for use in a project like this one.

Ekahau Ekahau is a product by a Finnish company that is built to locate commercial data such as products, carriers and drivers inside a city. They do not have a SDK that can be used for prototyping, and seem like an industrial solution. As such, it is not very useful to our project

Microsoft RADAR and CHOICE Microsoft developed a system that also uses WiFi localization. However, their system works slightly differently. Instead of relying on a database, RADAR is calibrated by taking precise measurements inside the building. These measurements allow the system to work more accurately, but adds an extra level of overhead. Moreover, the calibration needs to be repeated if the setup indoors should change marginally (i.e., when moving a closet in an office). Due to this extra layer of complexity and the extensive calibration required, these systems are only meant to be used indoors. They are therefore not very practical for this project.

2.3 Survey of Location-based Services

Most web sites and programs currently operate with a limited amount of information about the user. They can tell time, and signal when a certain event is scheduled. Although these services have been very useful until now, most people have increased their mobility. Mobile computers are all around us, disguised as cell-phones, PDAs or even watches. In order to be truly useful, another attribute needs to be added: the location.

Location-based services can loosely defined as programs or web sites that takes into account the location of the person using the computer. As covered in section 2.1, various technologies can be used to obtain the coordinates of the user. The program that we are building within the scope of this project will also be such a service. Before starting the design of the program, it makes sense to research the location-based services that are already in place.

Although location-based services are just starting to make an appearance, there already are major products that attempt to use location information to provide services to the user. These products vary in their interface: some are developed for cell-phone usage, while others are meant to be run from within the browser. This section will take a look at some of these services that are available today.

2.3.1 SociaLight

SociaLight is a very promising startup that allows users to perform the most fundamental activities. It lets one assign notes to locations that other users can look at. This is somewhat similar to posting virtual Post-It notes on a map.

Through these messages, information can be readily shared between people. When walking or driving through a neighborhood, the software that runs on your mobile phone can let you know about the posted notes for that area. It can filter the notes based on the types you want to see, and can show localized advertisements.

The system relies on loki for its location sensing, which is provided by Skyhook wireless (see subsection 2.2.1). The system is currently a prototype, but it is expected to be available on both cell-phones and computers. When run through a phone, it uses the GPS receiver that is built-in the phone. Wireless localization is employed when the page is accessed through a computer.

Another interesting feature is that it is compatible with most social networking sites (facebook, myspace and various blogging systems). It therefore makes the job of integrating location data into current applications very easy.

2.3.2 loopt.com

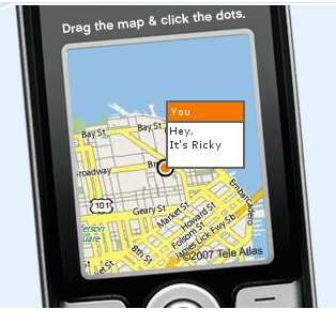

Figure 2: Screenshot from loopt

Loopt is a social networking site that is enhanced by location-sensing. Similar to most social networking sites, it maintains a list of your friends. The user has the option of being alerted when a friend comes nearby. Another feature is the ability to meet others in the area that match certain criteria.

It is currently supported by certain carriers (including Sprint and Nextel), and needs operator support to work and is available with a monthly fee. Since it uses the built-in GPS module in the phone instead of Wi-Fi networks, it needs to be supported by the phone.

2.3.3 Active Campus

The Active Campus is a system that uses the wireless networks that are widely available on college campuses to provide location-based services. The purpose of the project is both to develop these systems, and also to understand how such systems are used in real world educational scenarios, mostly to enhance student learning.

It's main feature is called ActiveCampus Explorer, and it uses a person's location to engage them in campus life. Although this is a very interesting project, it is not being maintained, and has not been updated in four years.

2.3.4 meetro.com

Meetro is a location-aware IM client. It lets one meet people that are in their vicinity neighborhood through instant messaging. It is compatible with most currently used IM systems (AIM, MSN etc..). However, one big disadvantage of Meetro is that it relies on the user inputting their own location manually. As such, it does not quite qualify as a location-based service as we have defined it, but it still demonstrates how these services can be useful.

2.3.5 Student Projects for PlaceLab

PlaceLab is a localization library that we have introduced in subsection 2.2.2. It was also used as a framework for a graduate class entitled "Location-aware computing" at the University of Washington.

These projects are very interesting application of the system. Some notable ones include the "Location-aware To-Do application" and the "Place Extractor: Translating coordinates into places." This latter project is interesting in that in transforms absolute, numerical coordinates on the globe to more "human" locations, such as "the library" or "the dining hall."

2.4 Web Services

One of the goals of this project is to explore the new technologies available. In the Web arena, one of the most important emerging technologies in the last few years was that of Web Services. They were therefore examined for the sake of the project. This section presents an overview of existing web services technologies in a theoretical way. A more hands-on approach will be covered in the implementation section 4.3.

2.4.1 Introduction

As described by the W3C, a web service is "a software system designed to support interoperable Machine to Machine interaction over a network." In more technical terms, it is a way of implementing code reuse over a network, through HTTP protocol.

When the computer systems emerged, it was instantly noticed that most of the code required for a program to run needed to be duplicated. Such common code was then placed in libraries, compiled files that could be shared between computers. When a program was invoked, part of the code could be fetched from the external file and executed. The advent of libraries improved software development on several fronts. Code that was used by several programs could be placed in a single library, allowing it to take less space and to make the job of updating it easier. It also made it possible for companies to market libraries. A company could write a code that performed a specific job efficiently (such as matrix operations, or GUI code), and other projects could license this library for use in their own machines.

Web services essentially transpose the same ideas found in libraries to a network (most often the Internet). When an application needs some data to be fetched or processed, it can send a request to a web service through the familiar HTTP protocol. The message that is sent contains the name of the service that is requested (similar to the function name in a regular function call), along with data that is needed for the function (similar to function

arguments). The server processes the given data appropriately and returns another message that contains the resulting data (similar to the returned value).

The intention of Web Services was to revolutionize Business-to-Business (B2B) communications. Currently the Internet can be seen as a collection of individual web sites offering their own services. The communication between them is almost non-existent, except for some of them like PayPal offering the possibility to make payments. The next step in the evolution of the Internet was seen to be more interoperable services. Web sites would be able to offer the information they possessed to others, making the information even more valuable.

There are currently two architecture that specifies how Web Services can be used. They differ in how the messages that go back and forth are composed. The Service-Oriented Architecture has a more rigorous definition of the interface, and the resulting types, whereas the REST architecture is more loosely defined. They both rely on XML to send and receive the data.

2.4.2 Service-oriented Architecture (SOA)

SOA Web Services have a very strict way of representing the messages that the server can received. This interface is published by the web service through a Web Service Description Language (WSDL) file. The file contains the name of the services that are offered, and what arguments each accepts and returns. The client uses this information to generate a message that conforms to the specifications. The message is sent to the server in a special XML format, called SOAP. Although SOAP originally stood for "Simple Object Access Protocol," the acronym was dropped for being misleading. The server receives the SOAP message, parses it appropriately and returns another SOAP message containing the result of the operation.

Another interesting feature of SOA web services is that they are "discoverable." The Universal Description, Discovery and Integration (UDDI) servers

act as a yellow-page of available services. This allows client to look up a certain kind of services or company that does the function that is needed. By downloading the WSDL file, the client can incorporate that service into its own framework.

The main criticism for SOA web service is that they are too complicated. There are tens of different standards that dictate how certain messages should be shaped, and how the protocol should work (e.g., for authentication). A typical web service book can easily contain thousands of pages.

2.4.3 Representational State Transfer (REST) architecture

Compared to SOA services, REST services are very easy to describe. They work along the same model, where the client and the server communicate by sending XML files through HTTP. The structure of the XML file can be arbitrary, and should be documented by the service provider. The client just needs to look at a sample file, and generate a similar one that will be sent to the server. The response will then be parsed by any of the XML parsers available, and the data can thus be used.

It is worth re-iterating that the reason why REST web services are seen as most successful is because of their ease of use. Although they are not as complete for more advanced jobs, they make the task of writing rudimentary services very easy. Most people foresee that REST services will almost surpass the SOA services in terms of acceptance[2].

2.5 Java Programming Language and supporting libraries

The language that is used for developing this application is Java. The choice for using Java was straightforward. For one, PlaceLab is written in it. It

[2]"Amazon has both SOAP and REST interfaces to their web services, and 85% of their usage is of the REST interface. Despite all of the corporate hype over the SOAP stack, this is pretty compelling evidence that developers like the simpler REST approach." *Tim O'Reilly.* `http://www.oreillynet.com/pub/wlg/3005`

is much easier to interface with PlaceLab if development is in the same language. We can then import the libraries into the project within Eclipse, and directly use the functions.

The second reason Java is a convenient choice for this project is because it has proven itself as a language suitable for large project development. By forcing the use of classes, it promotes good abstraction practices. It also runs on a virtual machine, it handles garbage collection, freeing the developer from the big burden of manual memory management.

2.5.1 Java Runtime Machine

For development purposes, I am using the latest Java Runtime Environment (JRE). At the time of this writing, that would be Java SE 6 (1.6.0_05). This Java version supports a lot of new innovative features. Some of them are as follows:

- pluggable annotations
- generics (Prototypes for data structures)
- Improved Web Services support (JAX-WS)
- Improved Java Binding of Objects to XML (JAXB) *used in Web Services*
- Scripting Language Support
- JDBC 4.0 support
- Java DB built-in Apache Derby database management system
- *Dramatic* performance improvements

Although not all of the features available in this version will be used, these are all big selling points of the new Java release. With the goal of exploring

new technologies in mind, I am considering experimenting with them simply to see if they are really as worthy as they are promoted.

Some of the novelties will be used extensively. These include the Java DB built-in database, libraries for using Web Services (such as JAX-WS and JAXB) and the latest JDBC 4.0 for interfacing with several database mechanisms.

2.5.2 SWT/JFace GUI library

The main part of this project will be a Graphical User Interface (GUI), with a very detailed set of specifications. The user interface needs to be intuitive, since it is going to be used on portable devices, such as cell-phones, smart-phones and PDAs. The biggest limitation to usability in this case is the size of the screen, which does not typically exceed 4" diagonally. Screen space being a scarcity, the GUI toolkit that I use needs to have a big library of widgets, and also allows the user to defines his own. Instead of using simple, HTML-style items such as text boxes and buttons, advanced controls like trees, lists and even browser controls are required.

A choice needed to be made between the three graphical toolkits available for developing Java applications. These are as follows:

AWT This is the first graphical library that was released in 1995. It was mainly designed to use the native toolkit of the Operating System, and therefore was fast, but not very portable. The look of the application was not the same between different platforms. Moreover, only a very small subset of widgets that are available in all the different platforms are supported (the "least common multiple" of widgets). The objects are therefore very elementary, and not very customizable. Currently, AWT is considered a legacy technology, and it is not widely used in modern applications, except perhaps in Applets.

Swing Designed to address AWT's difficulties, Swing is completely platform-

independent. This convenience comes at the expense of speed; all the widgets are drawn programmatically in Java, and do not rely on the OS at all. It is supported in the JRE, and does not require any external libraries. There are several large-scale projects, such as the well-known Bittorrent client Azureus, that use Swing as their graphical toolkit.

SWT/JFace Developed for the Eclipse IDE project (see 2.6), SWT attempts to complement AWT and Swing, by locating itself in the middle of the "Speed vs Look" debate. In SWT, the widgets that are provided by the OS are implemented as such, and those that are not available are drawn manually. This guarantees a very consistent appearance, and it also maximizes speed. The JFace framework allows the application of the Model-View-Controller pattern for SWT. It makes it possible to bind data structures to widgets, and takes care of updating the widget when the Model changes.

Considering all of the options, I decided to use SWT/JFace for this application. It has a very rich set of widgets, and it is proven to be usable in very large projects such as Eclipse. The JFace addition is also very convenient. Although it takes some time to learn about all the different classes and how to use them, once mastered the GUI "just works."

2.5.3 WSDL2Java converter and Ant task

In the traditional Web Service development process, one first starts out by describing the interface using the Web Services Description Language (WSDL) file (see section 2.4). This file is a specific XML Schema that contains the name of the methods, the connection point and port of the service, and the type of the arguments that need to be passed. In my case, the WSDL file has been generated using the WTP toolkit of Eclipse (see section 2.6).

Once the WSDL file is in place, the WSDL2Java converter program is used to generate stub Java classes. These classes contain all the methods that are

necessary to use Web Services in Java. The key methods that should contain the majority of the server-side code are left empty. It is up to the user to fill out these sections to get the server to work.

The WSDL2Java script that I used came with the Axis2 Web Services platform (see section 2.7). It can be executed from the command-line, taking the WSDL file as its first argument, and generating the package structure as a side-effect. There is however a better way to use this system.

The Axis2 library also comes with an ant task for automating this process. The steps necessary to get it to work are not very evident. One first needs to set up the *build.xml* properly to include the right libraries. Then the correct hierarchy of folders need to be created such that the output classes are considered as being with the source code.

2.5.4 MySQL Connector-J

As explained in 2.10, the MySQL database management system is used on the server-side to store the price information. Since the Web Services platform is somewhat convoluted (see 4.3), I needed a way to test the database connection without using Web Services. I decided to connect from my client to the server directly using a JDBC database connection. However the built-in JDBC library does not have out-of-the-box support for connecting to MySQL. The MySQL Connector-J library is needed for this purpose.

When the library is included in the classpath, the necessary driver is automatically loaded when the default JDBC method *getConnection* is called with "mysql" in the connection string. From that point on, the default JDBC methods such as *createStatement* and *executeQuery* can be invoked to run SQL statements.

2.6 Development Environment

Before boarding on a software engineering project such as this one, one needs to pick his tools. The development environment is probably the most important tool of all, since it is the programmer's interface with the compiler (or in the case of Java, the virtual machine).

Luckily, when it comes to Java development environments, the choice aren't that many, and they are all very attractive. Eclipse and NetBeans are both very compelling products, and they are both widely adopted. Since throughout my studies, I was mostly introduced to Eclipse, I picked it as the IDE for this project for purely personal reasons.

The Eclipse Development environment is the preferred Interactive Development Environment (IDE) for many Java developers. The latest version, Eclipse 3.3 Europa, is being used for this project. This program single handedly speeds up the development progress for the following reasons:

- The structure of the project is fairly complicated, with the project having a lot of dependencies on several jar files and libraries. It also uses a lot of different versions of tools that need to inter-operate. The project also contains a lot of classes and packages for a modular design and good software engineering practices. Although this ensures that the tasks are broken into smaller chunks of code, managing all of these files manually through compile-time options would be very unfeasible. One needs to be sure he is in the right directory for compiling it, make sure the classpath is set up properly, and go through the write-compile-run cycle several times. Writing Ant files would be preferable than handling the task manually, but would still be hard to maintain. Changes would need to be made to the *build.xml* file with every change, and the file would become unwieldy to manage in no time. Eclipse very conveniently takes care of all these development chores. The source code is maintained in the source directory, and the libraries can be added to the classpath using simple dialog boxes. Running the code is done

with the click of a button, and the compilation is done only if changes were made to the files.

- Eclipse has a very comprehensive debugging support. It allows the placement of breakpoints at problematic locations and the ability to watch variables until they get a certain value. Once the execution is paused, the developer has the ability to step through the code line-by-line, selectively entering and exiting functions. At any point, the variables in the scope can be monitored, and their values displayed. All these features make it possible to spot errors in a very short time.

- Eclipse has many plug ins for extending the out-of-the-box features. For instance, following well-known software engineering practices, I decided to use a version control system (VCS). These systems allow the developer to save the source files of the project in a central repository at several points throughout the development. It can be done at regular time intervals, after a certain milestone has been reached, or before tackling major changes. It is then possible to recover a previous state of the code, make a different change, eventually creating a different branch. CVS is the classical system among versioning systems, and is supported natively by Eclipse. However Subversion is a newer system that superseded CVS recently, and it addresses a few of its shortcomings. Being fairly new, Subversion is not natively supported by Eclipse, but can easily be added by the use of a plug in.

- Another helpful plug-in that I used was to the Web Tools Platform (WTP). It actually is a set of plug-ins for managing web development tasks and writing web applications. I used them when designing the web services server. As explained in 2.5.3, one needs to design the WSDL file. Having a graphical interface for designing XML files is a welcomed addition. Since the XML Schema is known by the WTP plug-in, a developer can use graphical objects for defining the web service

ports, functions, argument types and such by using boxes, arrows and text boxes. This speeds up the process, and makes it more foolproof, because Eclipse takes care of writing the XML file from the graphical representation, keeping the attention-related errors at a minimum.

2.7 Apache Tomcat and Axis2

This application will use Web Services as its data providing mechanism. In order to run a web services, one needs a server that supports the said protocols. The Apache Foundation provides an open source solution that comes in two parts.

Tomcat is used as the Application Server. It is entirely written in Java, and it allows one to run Java web applications written using servlets and jsp. It can either run independently, or it can be made to work with the original Apache server. In the latter scenario, the static web pages and binary files (such as images) are handled by Apache, and the Java-related dynamic pages are delegated to Tomcat. For the purpose of this project, no files will be published, so the main Apache web server was not needed.

To run web services, Tomcat needs to be augmented with Axis2, which is also developed by the Apache foundation. It is developed as a servlet application, and distributed as a file with the extension *.war* (which stands for Web ARchive, the counterpart of the *.jar* file for web applications). In order to deploy it, one copies the file to the *webapps* directory of Tomcat, which then takes care of decompressing the content in the right folder. The Axis2 system can then be accessed just like a web application through a web browser.

As an added bonus, the Axis2 project also comes with a WSDL2Java converter, as well as a Java2WSDL (that is not used for this particular project). These tools can be used directly within Eclipse thanks to the custom Ant task, also distributed with the package (see section 2.5.3).

2.8 Item Pricing services

In order to get the price information for various products, the project needed an online price database that also has physical store location information. This central database is to be accessed from the Personal Assistant. A search determined by the current location of the device will be conducted, and the result displayed in a map.

There are many web sites that provide the user with price comparisons. The prices are fetched from other merchants, and the result of the query is displayed. This convenient features allows the user to get prices from several web sites at once, making sure that the product is bought at a good price. Examples for these sites include `www.pricegrabber.com`, `www.mysimon.com` and `www.froogle.com`.

For the sake of this project, there are two additional requirements on the pricing service. It needs to contain prices for physical stores and not just online web sites. Shopping from web sites defeats the purpose of the Personal Assistant. We need to display the stores that are within close driving range from the user, so as to handle their errands. Secondly, the price information should be made available through a web service. That way the integration with the software would be made through the regular SOAP protocol, as described in section 2.4.

Unfortunately, we were unable to find such a web site or service. Noting that these two requirements were too strict, we decided to relax the second one. After some research, it was obvious that web services did not have as much acceptance as we were led to believe from all the hype. We decided that we could manually parse a web page (see section 2.9) and use the output as my own web service.

The requirement of finding a web site that lists the items in stores, along with their locations was also hard to satisfy. There is no standard for sharing this kind of price information. An experimental project developed at the University of Mannheim came very close, but still could not satisfy all our

criteria.

The aim of the shopinfo.xml project is to develop a standard that will allow the stores to provide their product information publicly, in an easily parsable form. This is done by placing an XML file entitled *shopinfo.xml* at the root of the web server so that it is accessible by going to the link `http://www.website.com/shopinfo.xml`. The price comparison services will use spiders for looking for the file, and if found will add the content to their database. Interestingly, this standard allows for physical locations to be specified as well. The downside is that most of the web sites that currently support the standard do not use this feature, and rely on selling online.

In conclusion, we werew unable to find a web service that provided the kind of information that was needed for this project. As the project could not continue without prices, we decided to manually obtain the price and store data by parsing the web site of grocery retailers, as explained in the next section.

2.9 Perl Scraping

The main purpose of this project was to find a web service that provided price information from different stores through a web service (see section 2.8). When such a service was not found, it needed to be created from scratch, and the data for it needed to be fetched from somewhere. The best option was then to parse several grocery stores' web sites using an HTML parsing script written in Perl.

Perl is a programming language widely recognized for its excellent text processing capabilities. It has a very powerful regular expression engine, and numerous built-in functions for separating, searching and displaying text. It also has an online library of modules that can easily be downloaded. These modules prevent the programmer from reinventing the wheel, and can add substantial amount of functionality very easily to the program.

Initially, our aim was to find a Web Service that would provide grocery price information to the public. After much exploration, we were disappointed to realize that no such service existed. The only other option was to create our own Web Service for this feature. In order to obtain the data that was to be served, we wrote a spidering script that browses the web sites of supermarket chains and collects the price of the items that are on sale. For the sake of example, the scraping script has been implemented for the Price Chopper web site. Repeating the same process for other supermarkets' web pages will be essentially the same job. It is therefore sufficient to do it once as a proof-of-concept application.

The script was entirely written in Perl, which facilitates text-based processing of pages. Moreover, the Comprehensive Perl Archive Network (CPAN) is replete with modules that are built to automate spidering. By consulting the excellent book "Spidering Hacks" by Kevin Hemenway, we found out that two of those modules were especially helpful.

WWW::Mechanize This module programmatically simulates a browser. Not only does it handle all the low-level HTTP protocol methods, but

it also has methods for clicking on specific buttons and links, and accessing the content of the current page. More sophisticated features include the ability to fill out form elements.

HTML::TreeBuilder Once the content is obtained through the *WWW::Mechanize* module, it needs to be parsed to get the price information about the products. This is achieved by the use of the *HTML::TreeBuilder* module. Its *look_down* method is very helpful for getting all the nodes and their inner *HTML* for the *DIV* tags that match a specific condition.

In addition to the modules used for scraping, some other modules were needed for inserting the information in the SQL database. The *de facto* module is *DBI*, which is similar in nature to the Java JDBC library. It has a standard interface that is independent from the database server that is used. Actually, the module itself is merely an interface, and the bulk of the logic is contained in separate "driver" modules, such as *DBD::mysql*. In theory, one can simply install this module using Perl's standard CPAN module, and start using it. However, our experience did not go so smoothly.

Our development system being a Windows XP, we installed the binary version of MySQL. As it turns out, *DBD::mysql* requires MySQL to be compiled from source code, because it uses the C client-side libraries to perform the logic. We therefore could not manage to install the module, without going through all the trouble of compiling MySQL from within Cygwin.

After some research, another module came to our rescue: *Net::MySQL*. In that module, all the logic is implemented directly in Perl, and therefore does not require any extra libraries to be present. Although more limited in its features, it still has most of the basic capabilities. That is why we ended up using this module for inserting the data collected through spidering into the database.

2.10 MySQL database

Databases are built for efficiently storing and retrieving large amounts of data. Since a web service that provided store information could not be found for this project (see section 2.8), we relied on collecting my own data through parsing (section 2.9). The dataset that we had was quite large, and it needed to be stored for easy access. The MySQL database was used for this purpose.

This application is very well-served as far as databases go. PlaceLab uses a native Java database called HSQLDB to store its beacon information. The client-side program uses the Java DB that is distributed with Java SE 6. In addition to these two, we had to pick a database to be used on the server, that was to store all the price and store information. This last database needed to be accessed by the web service. A multitude of client can be querying it at the same time, which requires it to be a very robust one. Neither of the two database servers cited could handle this kind of load. We therefore chose to use the database system that is the jewel of the open-source world, MySQL.

Not only has MySQL proven its value in the industry, but it also has proven its convenience for the smaller projects. Being completely free in every sense of the word, it is also supported on many platforms, including Windows XP. Installing it was a breeze, and the server was up and running in no time. After spending some time configuring the various users and granting the proper access codes, creating the tables was completed momentarily.

3 System Design

After introducing the individual libraries and technologies that were in this system, we can now describe the design of the application. Just as with any other software project, it is important to establish a few design considerations first. This section introduces the design goals of the project, goes over the brainstorming process and introduces the class hierarchy.

3.1 Design Goals

A software engineering project starts by establishing the expectations from the final product. Having these design goals throughout the project allows the project to move towards the right direction when decisions need to be made. This section introduces the major goals that should be considered when reading through the rest of this report.

3.1.1 Use of localization

This application is a location-based service, and needs a localization library as discussed in section 2.2. There are several trade-offs when considering the library to use. The library must be available on a wide-range of devices so that the development will not be repeated for every platform. The language used for writing the entire software must be compatible with the library used in order to avoid compatibility issues.

The usage of localization also affects the service that needs to be performed. The application must do something useful with the coordinates gotten through the localization library. Usefulness is hard do quantify

3.1.2 Limitations of mobile platforms

A location-based service is not beneficial if it is used from a fixed desktop system. If the address is known beforehand, a geolocation service can provide

the exact coordinates of the location. Since the location is not changing, the program wouldn't be of much use.

The platforms where these services really shine are mobile. Their portability implies that the coordinates changes, and that these coordinates can be used in more meaningful ways.

Designing software for mobile platforms is a challenging task because it comes with strict constraints. One of those constraints found on all mobile computing platforms is the shortage of screen real-estate. Most portable devices have diminutive screens that make the devices smaller and more energy-efficient. Yet the cost of the extra battery life is paid when dealing with user interactions. The design of the program should take into consideration the small screen and the usability of the final product must not lack any functionality.

Yet another constraint is the programming environment. Developing programs directly on mobile devices is difficult. It is typical to conduct the development on a regular workstation, test the code using a simulator, and then test it on the device itself. Luckily, the Java platform is widespread for mobile devices. The Java ME platform is deployed on all modern portable devices, be they cell-phones, PDAs or smartphones. Java is therefore an ideal language to work with.

In recent years, mobile devices have acquired general purpose processor that allowed them to run any user-defined code in addition to their firmware. A typical cell-phone processor's speed does not exceed 200 MHz, whereas that of a smartphone is in the 500 MHz range. Although these speeds seem sufficient, they are not so when dealing with computationally-intensive algorithms. We must keep these considerations in mind when choosing algorithms.

3.1.3 Provide a unique service

Combining the processing power of portable devices with their location information opens the doors to a multitude of interesting applications. These ideas can range from the mundane to the innovative. There also are a lot of services currently developed that address this same challenge. For this project, we aim to build a software system will be truly productive for its user. This goal of the project is the only one that is not technical, and therefore the one that is the hardest to measure.

3.2 Brainstorming for ideas

The development phase began by brainstorming possible uses for the system. In order to satisfy the goal introduced in subsection 3.1.3, we listed all the applications of wireless localization systems in a mind map format. The challenge was to find an idea that showcased the power of location-based services that was useful, and that was not already developed previously.

Mind maps were used when brainstorming for ideas. These diagrams use various shapes and graphics to engage the right hemisphere of the brain which is usually ascribed to creative thinking. The mind maps are usually drawn in a tree structure, where the root of the tree is found at the center of the page. Each main branch stemming from the center corresponds to a major area where location-based web services could be used. By the end of this process, we obtained the diagram shown in appendix C.

Some of the ideas discovered through the brainstorming process had already been implemented by start-up companies listed in section 2.3. Of the remaining possibilities, a viable subject was picked.

3.3 Project setup

Once the theme of the project was discovered after the brainstorming phase, the design process began. This section will explain how the software was

designed, and what methodologies were used for it.

3.3.1 General design points

Before delving into the details of the implementation, we will go over some general concepts used throughout the design process.

Design patterns are reusable solutions for solving a specific problem. While many patterns were used throughout the design, some of them need special attention. The Model-View-Controller pattern is regarded as the ultimate design method for graphical applications. It partitions the entire code into three logical blocks, each having its own purpose. More information about the Model-View-Controller pattern can be obtained in [GHJV93] The MVC pattern is also the role of GUI toolkit that we used, SWT/JFace (see subsection 2.5.2).

Partitioning the project into smaller pieces is the goal of the object-oriented programming paradigm. Each logical block is abstracted as an object that can be manipulated through several methods. The application of the MVC pattern discussed above is especially suited to being used with the object oriented programming. Each class will either act as a model, a controller or a view.

Java further enforces object oriented principles by introducing the concept of packages. Big projects like this one may contain hundreds of classes. Grouping these into logical subsets makes the code more readable and structured, and provides a more precise control over the visibility of classes; objects can expose certain methods only to other objects belonging to the same package.

All of these tools have been used to make this project as rigorous as possible. The rest of this section will detail the packages and their contents.

3.3.2 Model Classes

The Model classes are divided into three separate packages based on their purpose. The classes that mainly use logic written by the authors belong to the model package, shown on figure 3. These classes deal exclusively with the grocery information, and the various mapping functions.

A *TreeNode* is the main abstract class for dealing with the storage of items in a Tree on the GUI. Since it is an abstract class, it does not directly have any fields that lets it store any useful data. Instead, the children and parent fields are used to capture the relationships between the levels of the tree.

There are two subclasses of TreeNode. *GroceryItem* is responsible for storing all the information pertaining to each grocery items. As it can be observed on figure 3, it has the fields for the name, store and price of the item, as well as the latitude and longitude of the specific store. The *GroceryList* class does not contain much logic, and it is aimed to be the root of the tree (i.e., it does not have a parent).

The *GMapGenerator* class writes the HTML file that contains the Google Maps API page. It takes some coordinate values as arguments, and writes the appropriate Javascript code to the specified file. This file can then be displayed in the Browser element of the GUI.

The *model* package has a sub-package entitled *datastructures*. This package has three classes for modeling and solving the Traveling salesman problem when calculating the trajectory of the visit. The *Vertex* class represents a vertex in graph, and holds the coordinate information of each store. Since in our graph, all vertices are connected to each other, there are no explicit edges between them; all vertices are understood to have an edge with all other ones. The *Vertex* class also has fields that makes it suitable to be used as a tree. By merging a vertex and a tree node, we can keep the same class structure when transitioning from the graph to the minimum spanning tree.

The *MinimumSpanningTree* class takes a set of vertices, and calculates

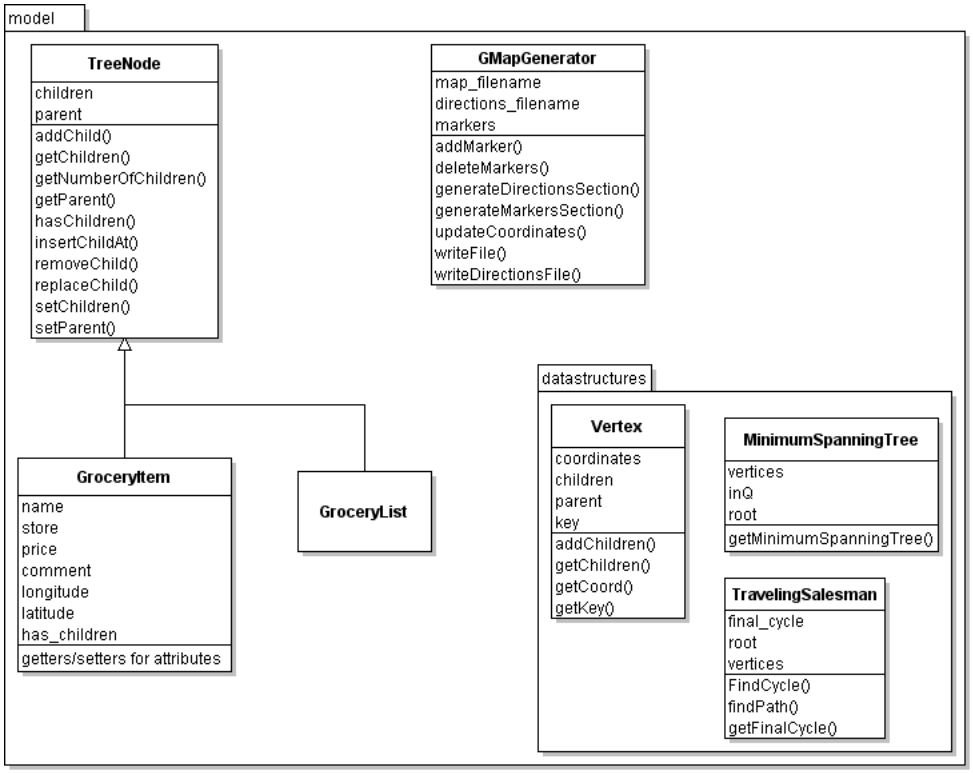

Figure 3: Structure of the *model* package

the minimum spanning tree from a specific starting point (the initial location of the user). The shortest route visiting all nodes can then be calculated from this tree by the *TravelingSalesman* class.

Another package that corresponds to the model logic is the *db* package (figure 4). It contains a *DerbyDatabaseManager* class that handles the storage of the grocery list using the built-in Java DB database. It essentially converts the GroceryList data structure discussed above to a set of SQL statements. The nature of the SQL queries can be guessed from the name of the methods (e.g., *getStoredGroceries* executes a SELECT query, while a *setStoredGroceries* executes an INSERT query).

The placelab package contains a single class, *PlaceLabConnector*, that acts as an interface with the underlying PlaceLab library (see figure 5). The

Figure 4: Structure of *db* package

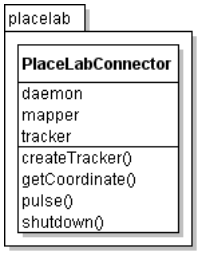

Figure 5: Structure of *placelab* package

PlaceLab installation contains tens of classes that need to be used simultaneously to get the current coordinate information. By hiding the calls through a class, the main code is greatly simplified. A simple call to the getCoordinate method returns an object that contains the current coordinates. It essentially implements the Facade design pattern.

3.3.3 View Classes

The View classes that handle the GUI related functions are enclosed in the *gui* package. These classes handle the user interactions and the display of information to the user. The structure of this package is shown on figure 6. The GUI code itself is in the *JFaceGUI* class. Although this class contains two dozen methods, they are not shown on the figure for the purpose of simplicity. They deal with the creation of the widgets (lists, buttons) and binding them with the controller classes (see section 3.3.4).

As suggested by its name, the *BackgroundThread* class runs in a separate

thread on the background and handles tasks that need to be run regularly. An example of such a task would be to query the PlaceLab library in order to obtain the current coordinates, and update these values on the GUI.

The rest of the classes are separated into their own packages depending on their purpose. The *labelprovider* package contains classes that describe how to display the data stored in the model classes in the widgets. As explained previously, all the information regarding the grocery items are stored in the *GroceryItem* object. The label providers take these grocery items, and extract the respective fields that will be displayed in the columns.

The GUI has one dialog box that allows the user to add and to edit the grocery items in the list. The *AddEditItemDialog* includes the logic for updating the model. The *actions* package is similar in nature to the dialog class. Each of the classes in this package performs a single task. They are called when the user clicks on a button in the toolbar, or clicks on a menu item.

The contentprovider package can be considered to be a hybrid view and controller logic. Each of the classes in that package are assigned to a widget. By overloading certain methods, the user decides what data will be displayed inside. For instance, the methods for accessing the web service for the grocery information and for fetching price are found in these classes.

3.3.4 Controller Classes

The project setup does not contain any strictly controller related classes. The job of reacting to the user input is handled exclusively by the JFace framework. When an item on the list is expanded, JFace calls the methods defined by the programmer to perform the correct action. The programmer is therefore relieved of the burden of explicitly dealing with user behavior.

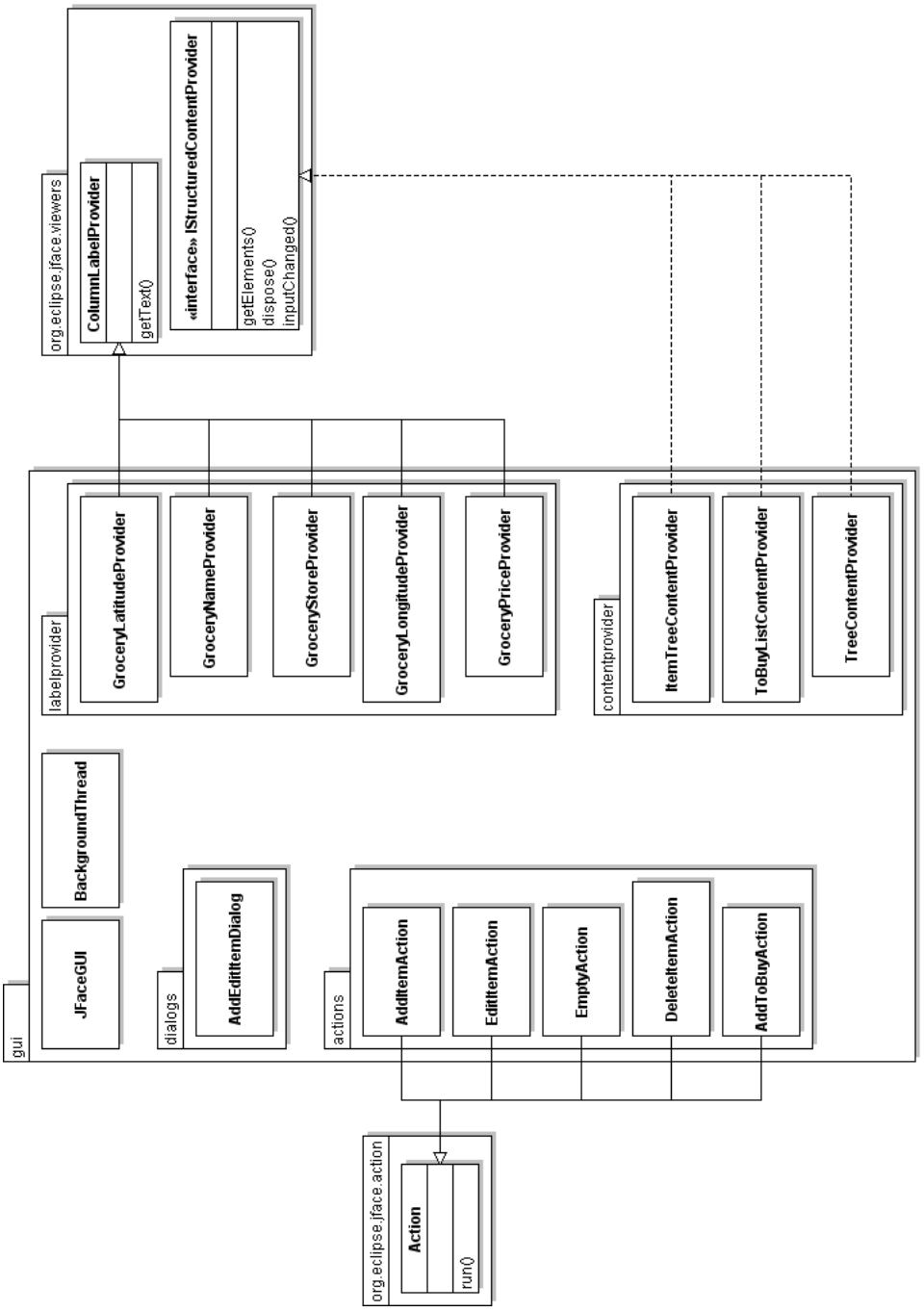

Figure 6: Structure of *gui* package

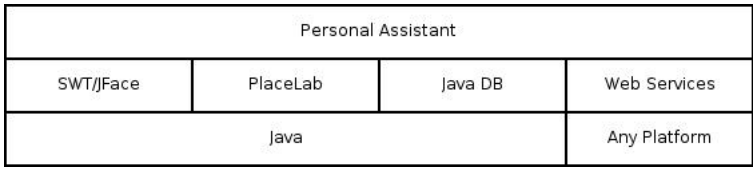

Figure 7: Block Diagram of the system

3.3.5 Block Diagram

With all the classes defined, it makes sense to look at the big picture block diagram of the overall system (figure 7). On this diagram, each block corresponds to a section of the program, and is dependent on blocks at the lower levels. At the base, of the entire system is the Java virtual machine. By definition, the web services are platform independent, and can therefore be implemented in any language (including Java).

At the second level, one can observe the different libraries and interfaces that are used. The location information is provided by PlaceLab, the local storage facilities are available through Java DB and the GUI toolkit used is SWT/JFace. The web services framework is also at this level, and is communicated with through the Internet. The Personal Assistant software itself uses all of these technologies.

4 Implementation

Having outlined all the different components of the system, this section will explain how they are put together to make the Personal Assistant application.

4.1 Graphical User Interface

4.1.1 Tabbed client GUI

Using the SWT/JFace toolkits, we designed a GUI that uses a tabbed design allows for a more efficient use of the screen space. The tab names can be designed using buttons so that they can be clicked with a stylus pen. The four tabs are labeled: Map, Estimates, Items and Stores.

Estimates The simplicity of the estimates tab (figure 8) can be misleading. It only contains two labels, but those labels show the current location of the user as obtained when the PlaceLab library is queried. The labels are updated every second in order to track the movement of the user.

Items The Items list is where the grocery items are stored (figure 9). The user adds each item that is needed to the list using the Add Item dialog box. Price and store data can optinally be included with the item to act as a reference. They can be used to compare the price quotes that are obtained from the server, to see whether the deals are really worth it or not.

Once the initial list is formed by adding all the items that are needed regularly, the Personal Assistant handles the task of finding the items in the local stores (as explained in section 4.3). The user needs to expand the item that will be looked up by clicking on the plus icon to the left of the item. This initiates the lookup, and displays the result of the query as sub-elements. Each of these items corresponds to a deal from a local store.

After browsing the list for the items that are wanted, the user selects the ones he is interested in. By clicking the “Update To-Buy” button on the

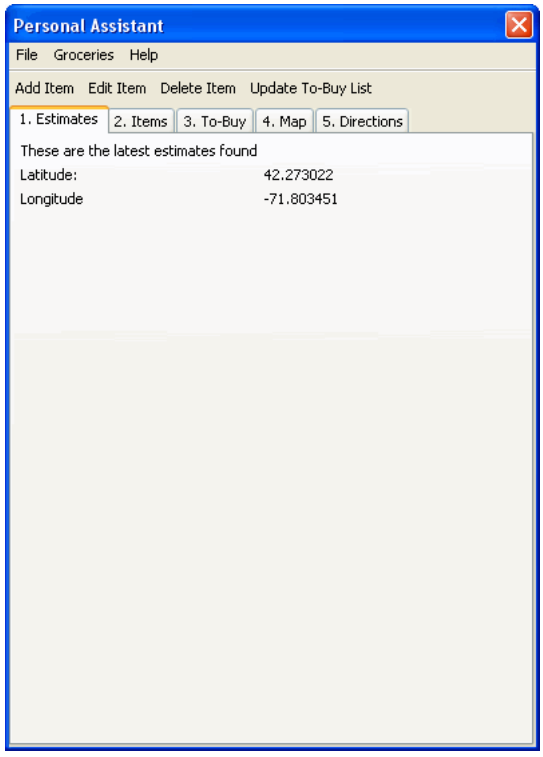

Figure 8: Estimates Tab

toolbar, these items are committed to a separate To-Buy list found on the next tab.

To-Buy List The To-Buy list allows the user to review the items that will be purchased. Only the items that will be attended to immediately are shown on this list, along with the location of the store (figure 10).

Map Once the To-Buy list is confirmed, the next step is to click on the Map tab (figure 11). This tab shows the location of the stores using the Google Maps API (see section 4.1.2). Each marker corresponds to a store that will be visited during the routing phase. The user has the full capabilities of Google maps: he can zoom in and out, and can even go to the satellite view

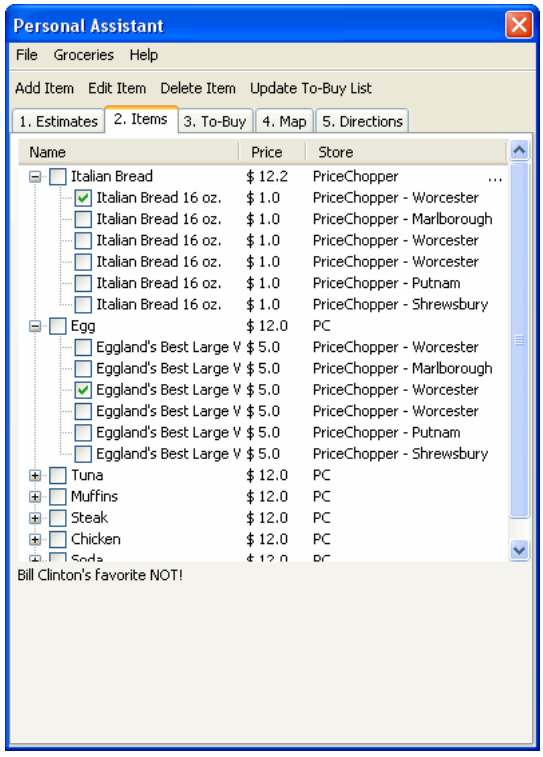

Figure 9: Items Tab

to see real pictures of the stores taken from the satellites.

Directions The final step in the procession is routing. By clicking the Directions tab (figure 12), the GUI starts to solve the Traveling salesman problem (see section 4.2 for details). The result of the computation is displayed on another map, and the route is shown by highlighting the route.

4.1.2 Mapping Facilities

One of the most exciting innovations in the recent years was the introduction of powerful client-side Javascript capabilities in the browsers, which opened the door to a multitude of applications. The Google Maps API is one such application. It allows the inclusion of dynamic maps to any web page. The

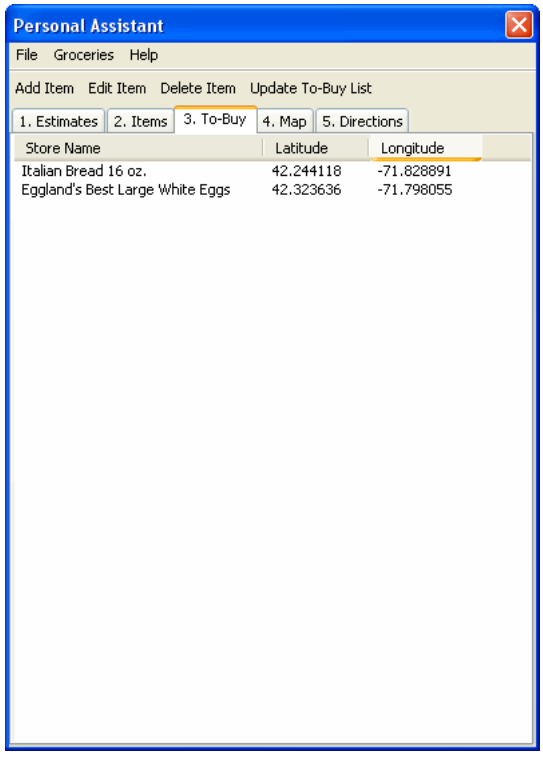

Figure 10: To-Buy list tab

user can move the map around and zoom in and out just like a real map. Furthermore, the API can also display satellite imagery instead of the map, making it a truly interesting application.

Due to its wide availability, the Google Maps API was used for mapping the store locations and displaying the directions. We did, however, run into several obstacles concerning the licensing of the API which is intended to be used by web sites and not standalone applications. There are no ways of displaying the maps through special map widgets, and there are no native Java bindings (although there are a few projects that attempt to make such a binding, none of them are production quality).

The final solution was convoluted and far from elegant, but worked exceptionally well. We added a Browser object to the Map and Directions tabs.

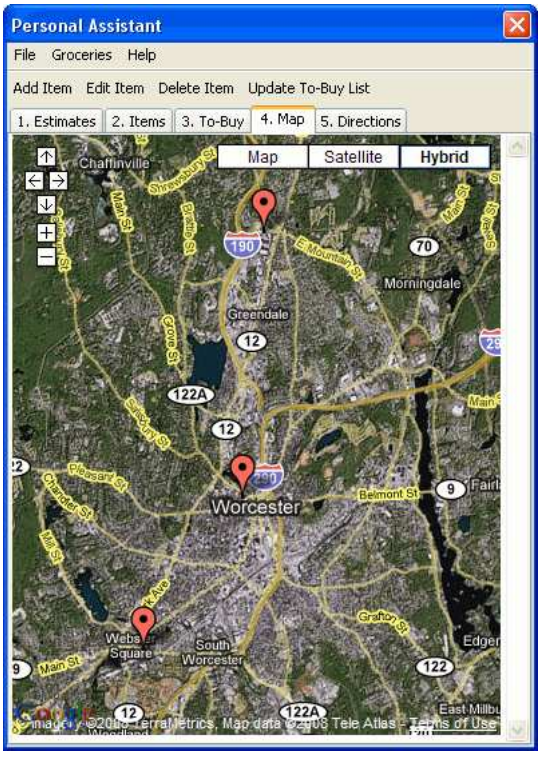

Figure 11: Map tab

When the tabs are made active, the *GMapGenerator* (see section 3.3.2) writes a file that contains pure client-side Javascript code. The Browser object is then instructed to display this file. From Google's perspective, this is just like looking at an offline page.

4.2 Traveling Salesman problem

Having a list of the best deals on certain items and the location of the stores where they are available is no doubt convenient. Yet it still leaves out some decision-making on the user's side: which store to go to first? In order to relieve the user from this extra burden, we implemented a solution of the traveling salesman problem (TSP). For this application, the problem takes the form of finding the shortest route (cycle) that visits all the stores in the

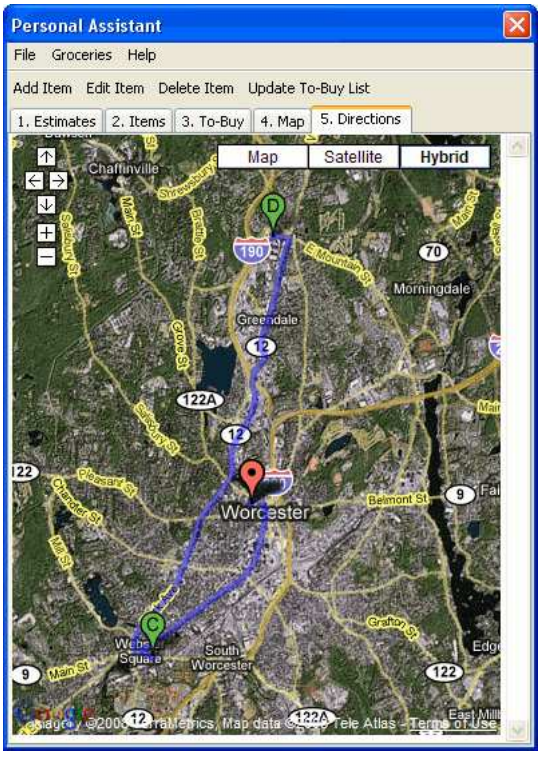

Figure 12: Directions tab

area. The shortness is measured by the distance between the stores.

The TSP is known for being NP-hard in the number of vertices. Based on the design decisions concerning the feeble processing power of portable devices (see section 3.1.2), we needed to find an appropriate way for solving this problem. Luckily, we could use some approximations and greedy algorithms to reduce the complexity of the problem. The number of stores that a user will want to visit in any given day is another factor that helps to simplify the algorithm: realistically, nobody would visit more than 10 stores.

In this implementation, the vertices of the graph in which we are trying to find a cycle correspond to coordinates on a sphere. By simplifying the reference object from a sphere to a plane, we can make use of the triangle inequality. The shortest distance between two points is a straight line, and in

a triangle on the plane, the sum of the length of two sides will always be larger than the length of the third side. Making this assumption allows us to use the Minimum Spanning Tree (MST) of the graph in order to find the shortest cycle. Without the triangle inequality, a polynomial-time approximation does not exist unless $P = NP$.

Out of the several possible implementation of the MST, we used Prim's algorithm, the details of which can be found in [CLRC01] . For the purpose of this section, it is sufficient to know that this algorithm yields the MST of a graph in total time $O(E \lg V)$ and V is the number of vertices (stores) and E is the number of edges (in our graph, all vertices have edges with each other). Once the MST is found, the shortest path can be found in $\Theta(V^2)$ time.

4.3 Server-side: Web services

The application is designed using the client-server model. The GUI running on the mobile platform constitutes the client-side. Although most of the processing is done on it, no data is stored. The evident advantage of this is in terms of storage space. Portable devices do not have much storage space on them, and they are therefore not very appropriate for storing a lot of information. The server on the other hand is a faster computer. It may have virtually unlimited amount of storage space thanks to server farms that are dividing the load to several computers.

The other advantage of delegating the data to a server has to do with synchronization. Having the price information locally would speed up the access time, but that data needs to be updated regularly in order to reflect the latest set of data. If an update is missed for any reason (i.e., unavailable Internet connection), the user may be presented with wrong kind of information. The client-server model is therefore beneficial for this program.

4.3.1 Price-providing web service

Initially, the intention was to find a web service that offered price and store location information. After much research (see section 2.8), such a service could not be found. Not only were there no comparison shopping site that offered such a service, but even the grocery stores themselves did not publish their weekly pamphlets through the SOAP interface.

4.3.2 Custom web service

We decided to create our own service just for the sake of this project. This job had two parts to it. There was first the step of gathering the data was going to be stored. The items that were on sale and the store location had to be obtained and saved in a database. Once the data was available, it needed to be offered to the client.

Data collection The information about the deals were gathered by parsing the web page of a certain grocery store. For the purpose of data collection, several Perl scripts were written (see section2.9). This step has been performed for the Price Chopper as a proof-of-concept, but can be repeated for any other web site.

The Perl scripts are responsible for fetching the HTML page, parsing the tree structure and extracting the prices and the kinds of deals available. The process is repeated for each store, and a geo-location lookup is performed to convert the addresses of the stores into absolute coordinates.

Once collected, the data is stored in a MySQL server (see section 2.10) through the use of INSERT queries. The database server speeds up the storage and access times because it is optimized for quickly fetching and indexing the data. Furthermore, almost all programming languages (including Java and Perl) have interfaces to the Open Database Connectivity (ODBC) API for easily manipulating data in a database.

Data publishing Once the data was obtained, and inserted in a database, it needed to be served by a web service. We consulted several books on the topic, including a few especially oriented for web services with Java. For simple tasks, the steps needed to follow seem trivial. One creates a WSDL file that describes the interface of the web service to the outside, then uses a WSDL2Java converter tool (see 2.5.3) to convert it to Java stub classes, and finally modifies the classes to include the logic. We went through this sequence with only minor obstacles, and created a simple web service that takes two strings as an input, and returns their concatenation.

Equipped with this background information, we went tried to generate the actual, full-blown web service. Yet we got stuck on the very first step, that of creating a WSDL file. The structure of the WSDL file was not clear as to how to define arguments of the service, since in this case they were not simple types like strings or integers, but objects and arrays of objects. Since WSDL files are by design platform-independent, one could not use Java object and classes in that file.

After some research and diving into another, more recent book, [Han07], we discovered that we weren't the only ones to think that designing web services was easy only for simple tasks, but overly-complicated for real-world applications, insomuch as they were completely redesigned with the latest release of Java.

Collectively called the "Java Web Services" (JWS) platform, there are four new technologies that make up an easier model for web service development. These are JAX-WS 2.0, JAXB 2.0, WS-Metadata 2.0 and Web Services for J2EE 1.2 (WSEE). These different modules are very new, and were first released with Java EE 5 and Java SE 6.

With all these new standards to master, and only limited time to the deadline, it became clear that web services weren't a feasible solution for this project. The lacking adoption in the industry is yet another proof that the standards are more complicated than they should be.

The alternative to SOAP services, REST-based web services, are becoming more popular and widespread due to their ease of use. They work by transferring XML files that do not have to abide by any standards. The server and the client are responsible for generating and parsing the XML file. As a result of this increase in flexibility, quicker development is possible with REST services, making them a promising standard for business-to-business communications.

4.4 Server-side: Database only back-end

Seeing as the Web Service alternative was not viable, the other remaining solution was to directly use a database connection from the client. We used the MySQL server for this case. This approach also has its pros and cons.

One big disadvantage is that it does not allow for any business logic to be incorporated on the server side. Database queries use the SQL language, which is very powerful and efficient for accessing and modifying the data. However its programming capabilities are severely limited, and does not go beyond that of arithmetic or simple string manipulation. If we had used a Web Service interface, we would have been able to incorporate code to do anything that a full featured programming languages allows one to do. Moreover, web services are language and platform-independent. The logic could have been implemented in the language that would best accommodate the job. A service that deals with text strings could have been written in Perl, while a service that did a lot of numerical computation could have been written in C with the appropriate libraries.

Another disadvantage of using databases surfaces when dealing with the security of the system. In general, it is usually not a very good idea to provide third-party users accounts to the database. Although MySQL supports a very sophisticated access granting system, the entirety of the server is still exposed to the outside world. On most systems that adopt the client-server model, a structure called an *n-tier* or *multi-tier* structure is used (see figure

13 for an example of a 3-tier architecture). The basic premise of this setup is flexibility and security.

The system is flexible because the whole complexity of the system is hidden behind one computer that works as an interface. Any number of additional servers can be present behind that. A load-balancing server can route the traffic to a number of other servers, preventing the connection to stall due to all the users accessing one of the servers.

In terms of security, this architecture relieves the data tier from providing the security. Instead, the logic framework can work on securing the system by checking for access privileges during authentication. The logic tier is usually handled by application servers (like Apache Tomcat) that are built to run on security-sensitive applications.

In our structure, the database needs to check the password of the application and give them the appropriate privileges. For instance, the client-side program only has read-only access to the price data, whereas the Perl script has to have write access when updating the price values. These access rules can be implemented in MySQL through the privilege granting mechanism. It is nevertheless not an ideal implementation.

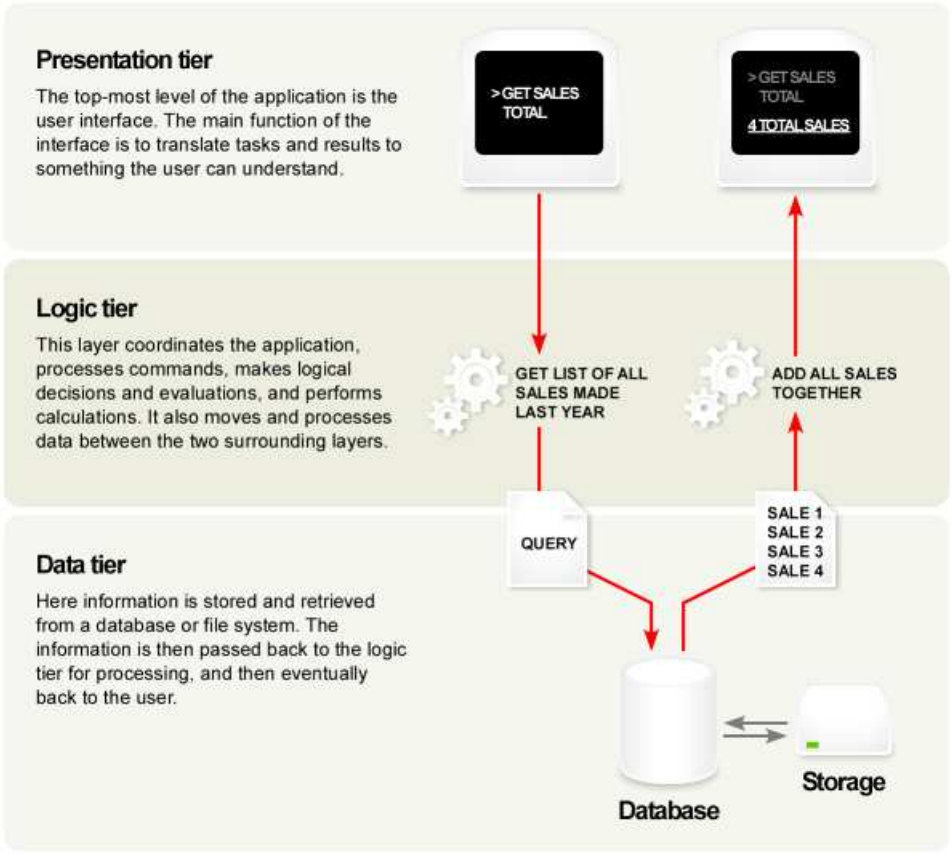

Figure 13: Example three-tier application

5 Evaluation/Testing

The application once completed, we went back to the design goals established in section 3.1 to check if they have been fulfilled. As a reminder, those goals were

- Use of localization
- Mobile Platform
- Provide unique service

Whereas the first two of these requirements are technical in nature, the last one is more subjective. The evaluation of the system will therefore be done in two stages. The technical characteristics will be looked at first, and the usability testing will be done by a third person.

5.1 Technical testing

5.1.1 Platform dependent specifications

We explained in section 3.1.2 that the program running on a mobile platform introduced many constraints. Most of these have been addressed by the implementation. For instance, the entire user interface has been made such that most of the screen is used to display data; only the three rows at the top are used for user interactions.

The program itself is kept very lightweight in order to reduce the load on the processor. Most of the computations are done on the server side, except for the routing algorithm (see next subsection). The data is also exclusively stored on the server. Only the regular list of items to buy is stored on the

5.1.2 Algorithm implementation testing

The traveling salesman algorithm is at the heart of the routing function. When the user selects the stores he needs to visit, this algorithm is respon-

Number of Vertices	Execution time (ms)
10	10
100	10
1000	110
5000	2173
10000	8512
15000	18887
20000	33538
25000	53968
30000	76591
35000	107064
40000	139380

Table 1: Complexity tests for the Traveling Salesman algorithm

sible for finding the shortest route that visits each point. As explained in section 4.2 and in [CLRC01], we use an approximation that uses the triangle inequality, and therefore the complexity of the algorithm should be $\Theta(V^2)$, where V is the number of vertices. In order to test this value, and to evaluate how well our own implementation works, we ran some benchmark tests. We ran the same algorithm in an isolated test bench (i.e., without running the rest of the GUI) by varying the number of vertices. The values for the coordinates of each vertex was obtained randomly using Java's native *java.util.Random* generator. The result of the test is shown in table 1. A graph of the same results can be found on figure 14.

The graph shows that the complexity grows quadratically. Although this is still not perfect, it agrees with the theoretical results. It is also worth noting that the algorithm uses a greedy approach, and therefore is not guaranteed to give the optimal route. Nevertheless it is optimum enough to be run on mobile platforms even for large number of stores.

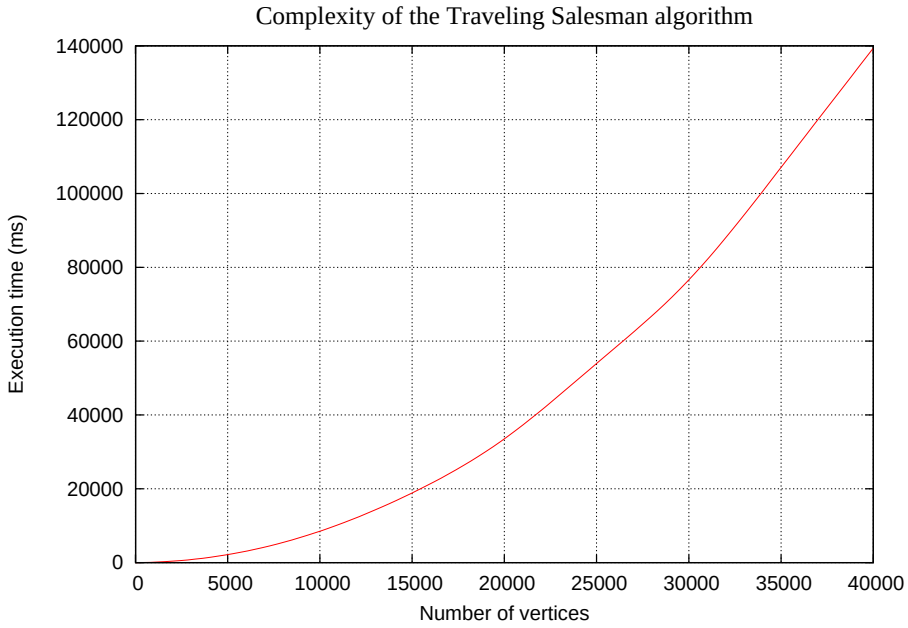

Figure 14: Time complexity of our implementation of the Traveling Salesman algorithm

5.2 Usability testing

The target audience of this project being the common shopper, its usability should be tested by somebody who does not have any knowledge of how the system works. Ideally, the person should not have an understanding of the various technologies employed, such as localization, the database back-end or even Google Maps.

We found such a volunteer to perform the tests. He is a senior undergraduate student at WPI who is majoring in biochemistry. Although he uses computers regularly, his usage does not go beyond that of checking his e-mail and composing documents with basic office tools. He therefore does not have any programming background. He is, however, truly interested in the system and what it had to offer. A test session was set up and structured in the following way.

The person was briefly introduced to the purpose of the program. This introduction was very superficial, and did not cover any directions on how to use the application, but it instructed the person as to what it was supposed to do. Later, a three sentence help text was shown. This text was meant to be very short so that it could be placed on a "Welcome" dialog box displayed when the application is first started. The interactions of the person with the system were then observed.

It took the person merely three minutes to understand the process. The numerals on the tabs were especially helpful in guiding the user. He understood that he had to start from the first tab, and move on to the fifth. Most of his time was spent on the second tab, that deals with the creation of a base list for the groceries. One source of confusion was the expanding list items. He failed to realize quickly that upon expansion, the lookup was performed with the database. Another obstacle was that of adding items to the "To-Buy" list using the toolbar item. The user had to go back and forth between the tabs until he realized what needed to be done. The relationship between the name of the third tab ("To-Buy list") and the label of the button ("Update To-Buy list") was a good hint.

Once the To-Buy list was formed, the rest of the steps followed swiftly. The person first looked at the map that showed the location of the stores, and finished up by looking at the directions tab.

All in all, the system was shown to be user-friendly. The learning curve for the program is under five minutes with only minor explanations. There are, however, several points that may need some improvement. For instance, the lookup of the items may be performed on a different screen, instead of expanding list items. Although this would be clearer to the user, that approach would also be more costly in terms of screen space. It is therefore a trade-off between usability and functionality. A status line that guides the user's actions would also be speed up the interactions. The software would then be similar to a wizard.

6 Future Work

Although we feel that the design goals we set initially were largely achieved, we believe that the program can still be improved further. Several interesting features could not be explored due to the timing constraints and several technical hurdles. These features range from quick improvements of the current system to larger ideas that may be interesting.

6.1 Improvements to the current system

The only major improvement to the current system would be the implementation of the web services. We strongly believe that they are the ideal medium to implement a client/server structure such as this one. Furthermore, once the service is in place, the data can be used for a variety of other applications with minimal effort. The platform independence of the system would allow web applications to access the price information just as desktop or mobile applications.

In section 5, we have found several points where the current graphical interface was lacking. The GUI may need some minor improvements to make it more user-friendly. For instance, the directions tab may be modified to show the textual, turn-by-turn directions already supported by the Google Maps API. Text labels on the tabs can be replaced by graphics to make them more descriptive and visually appealing. Contextual help messages may be added to guide the user throughout the process.

The underlying logic of the system can also be improved to make it better more intelligent. Currently, the user needs to browse through the available deals for the selected items, and make a judgment as to which one to buy. When making this judgment, the user needs to consider such factors as his time availability, the distance to the store and the attractivity of the deal. Ideally, these decisions can be made by the software itself, further reducing the burden on the user.

6.2 Follow-up ideas

One of the latest trends in the Internet is the advent of social networking. With sites such as Facebook.com or MySpace.com, people can interact with each other in completely new ways. A possible idea that may be worth investigating would be to create a companion web site to this software that will allow the users to input their own price information and create a kind of social network around grocery shopping.

The working of the software can be very straightforward. When a shopper finds a deal in a store, he will input the price and location of the item in the database. These user submitted data will complement the data that is coming from the web service. Once the database is updated, other users can see the localized deals on their Personal Assistant program just like the regular information. The system can be made more dynamic by allowing users to rate the deals, and flag them as expired.

Another path that is worth following is to expand the coverage of the system to more than just groceries. As of now, the software works only for grocery store. However, the location information can be used and applied to any kinds of stores. These may be technology stores, department stores or even hairdressers and gas stations. Since the code has been developed with extensibility in mind, the utility of the system can be easily increased.

7 Conclusion

Working on such a large-scale project in such a short amount of time taught us a lot of things, one of which is the importance of software engineering principles when working on big projects. Object oriented design made the development of this product run much smoothly. It may not have been possible to obtain similar results using C.

The importance of importance of code reuse and abstraction was also made apparent. We used a lot of different technologies, and tried to reuse libraries as much as possible. When working on integration projects like this one, it really does not make sense to duplicate the effort of re-writing a part that has already been implemented. The point of the project should be to make it such that the value of the system is larger than the sum of its parts. In other words, what you do with these many blocks is more important than implementing them yourself.

One final conclusion is regarding the state of the Web Services. We strongly believe that they are harder than they should be, which is hampering their adoption in the industry. Although they were advertised to be a revolutionary technology that would reshape how business-to-business logic was going to work, we found that they did not live up to the expectations. After experimenting with the technologies, I found out that the main reason behind the lack of adoption is in the difficulty of developing them quickly. SOAP-based web services are inherently hard to work with, which makes other alternatives such as REST-based services more attractive.

References

[Act] Explorations in community-oriented ubiquitous computing. `http://activecampus.ucsd.edu/`. Accessed March 17, 2008.

[CLRC01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Stein Clifford. *Introduction to Algorithms*. The MIT Press, 2001.

[GHJV93] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707:406–431, 1993.

[Han07] Mark D. Hansen. *SOA using Java Web Services*. Prentice Hall, 2007.

[HC04] Kevin Hemenway and Tara Calishain. *Spidering Hacks*. O'Reilly, 2004.

[Knu92] D.E. Knuth. Two notes on notation. *Amer. Math. Monthly*, 99:403–422, 1992.

[Loo] Loop - Your Social Compass `http://www.loopt.com`. Accessed March 17, 2008.

[Mee] Meetro.com `http://www.meetro.com`. Accessed March 17, 2008.

[New02] Eric Newcomer. *Understanding Web Services*. Addison-Wesley Professional, 2002.

[Plaa] Place Lab Hardware Compatibility List. `http://www.placelab.org/toolkit/hcl.php`. Accessed March 17, 2008.

[Plab] Projects using PlaceLab `http://www.placelab.org/projects`. Accessed March 17, 2008.

[Sky] SkyHook Wireless. `www.skyhookwireless.com`. Accessed March 17, 2008.

[Soc] Socialight.com. `http://www.socialight.com`. Accessed March 17, 2008.

[WCL+05] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture*. Prentice Hall, 2005.

[WiF] Wi-Fi Map of the World. Troy Dreier. `http://www.wi-fiplanet.com/news/article.php/3723366`. Accessed March 17, 2008.

A PlaceLab Installation under Eclipse

This section will show you how to import the Place Lab source code in Eclipse, both for using in a different project, and also for running. These instructions have been tested on Eclipse 3.3 running on Windows XP.

1. Download the PlaceLab toolkit from `http://www.placelab.org`. At the time of this writing, the most current version is `placelab-win32-2-.1.zip`

2. Under Eclipse, create a new project by selecting "File > New > Java Project," and name the new project "PlaceLab" (see figure 15).

3. Click on "File > Import," and under the "General" heading, choose "Archive File." After pressing next, select the download archive file using the browse button, and pick the just created PlaceLab project in the "Into folder" textbox (figure 16) . Click Finish to import.

4. Once the import is finished, you need to move the contents of the "src" folder under "placelab-win32" to Eclipse's default "src" folder. This can be done by dragging and dropping all the Java files.

5. The next step is to configure the build path. Right-click on the newly created project and go to "Build Path > Configure Build Path." The JAR files that came with PlaceLab need to be included as dependencies. Under the "Libraries" tab, add each of the items shown on figure 17 using the "Add JARs" and "Add External JARs" buttons. The SWT related libraries can be found under the Eclipse's plugins directory.

6. One the same screen, click "Add Class Folder" and include the "native" folder that came with PlaceLab (see figure 18).

7. You can now right-click on any of the demo applications (for instance org.placelab.demo.apviewer.ApViewer and run it by selecting "Run as" > "Java Application."

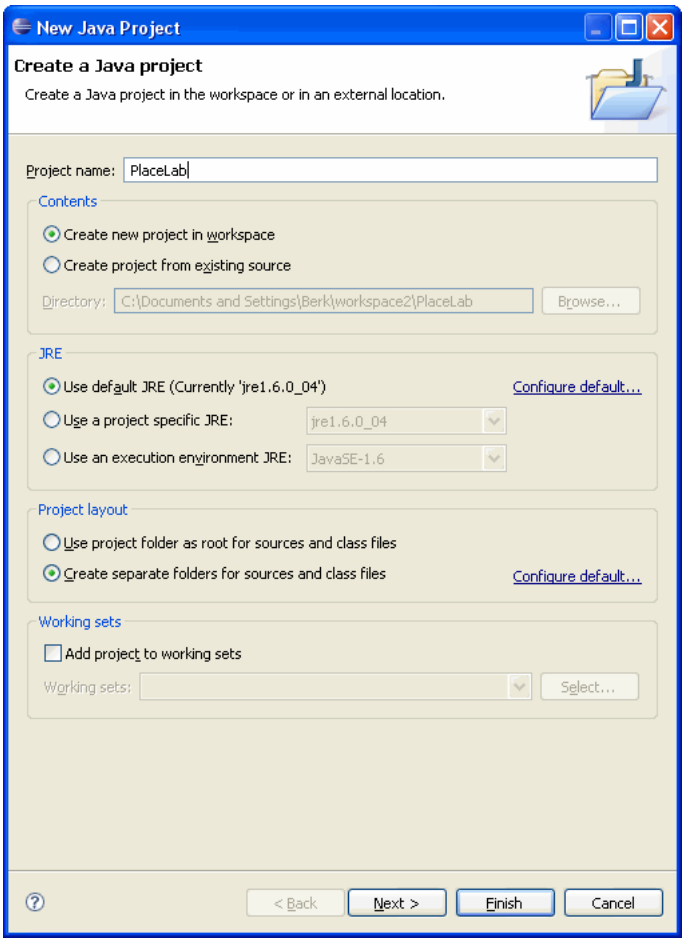

Figure 15: Creating a new project

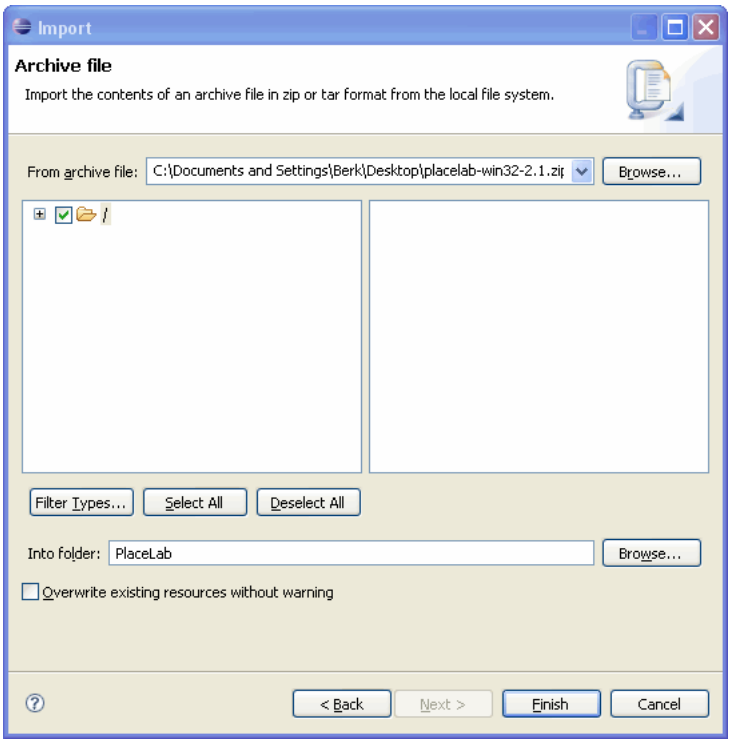

Figure 16: Importing the ZIP file

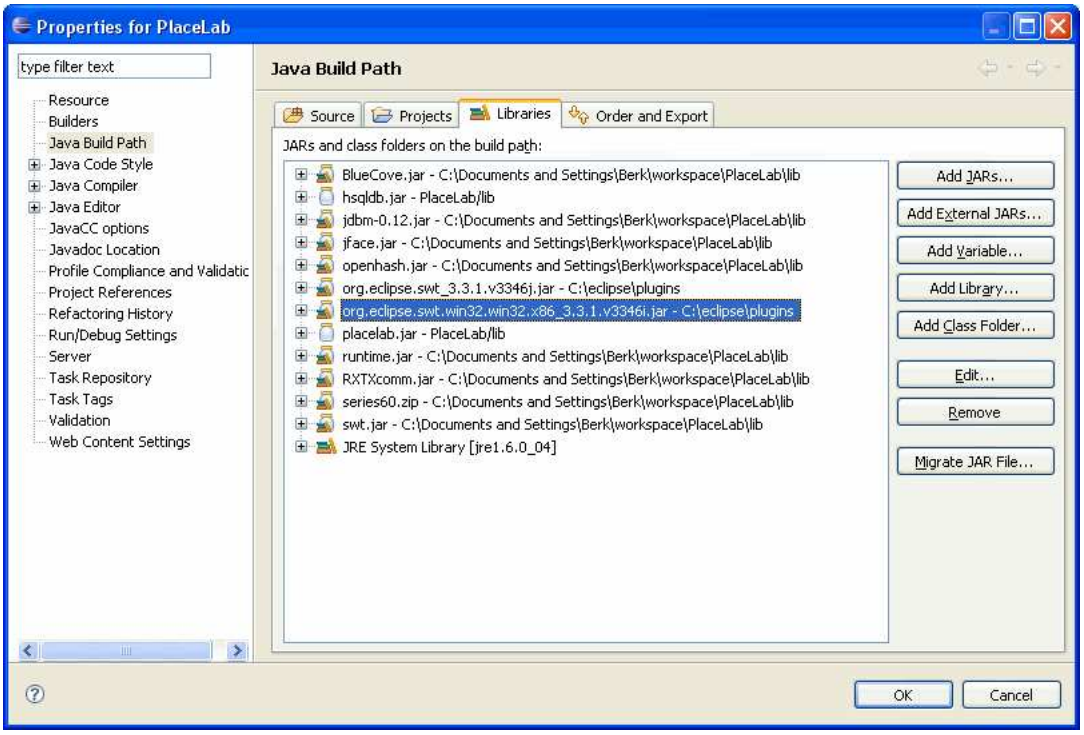

Figure 17: Setup the Build Path

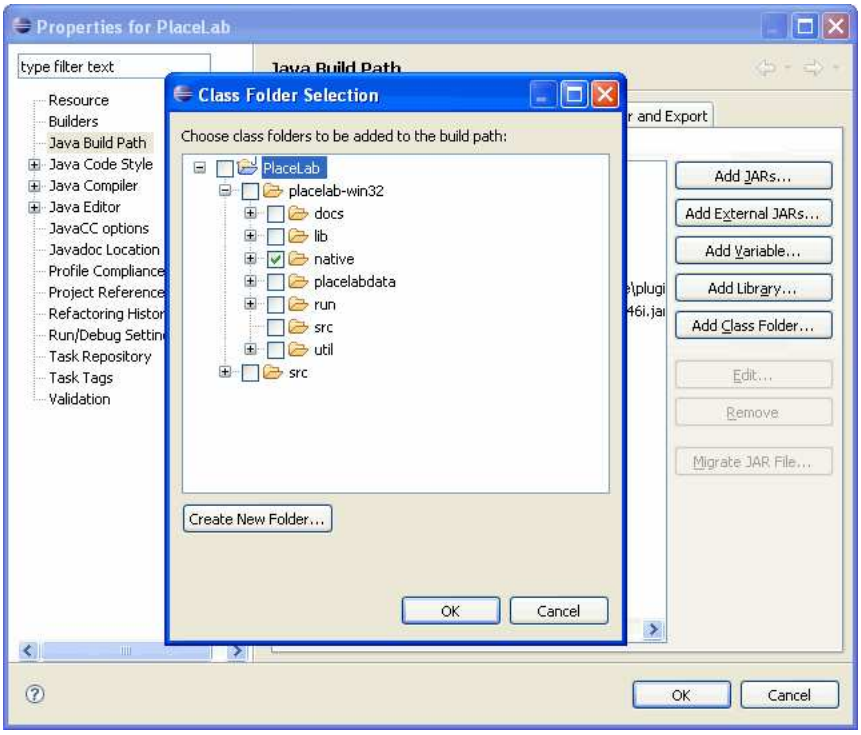

Figure 18: Setup the class folder for “native”

B Running Personal Assistant under Eclipse

This section details how to run the Personal Assistant program within Eclipse for either running standalone or for further development. Since the file has been packaged using Eclipse's export function, the user only needs to do an import to recover the state of the project.

1. Open Eclipse, and follow the steps outlined in appendix A to import the PlaceLab library.

2. Go to "File > Import" and pick "General > Existing Project into Workspace"

3. On the next screen, activate the "Select archive file" radio button, and click browse to locate the `personalassistant.zip` that is found in the accompanying CD, under the Source directory (see figure 19).

4. Once the project is loaded, right-click on the JFaceGUI class under the package `org.personalassistant.gui` and select "Run as > Java Application."

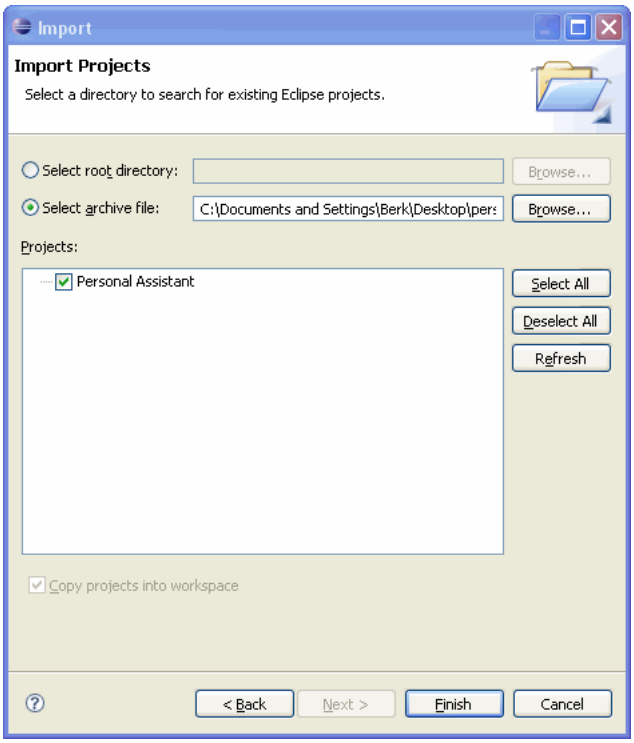

Figure 19: Importing the project

C Brainstorming mind map